# MITIGATING SPARK STRAGGLER TASKS FOR ITERATIVE APPLICATIONS BY DATA RE-PARTITIONING

BY

BO TENG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Roy H Campbell

# Abstract

Many of the data science applications nowadays feature large datasets and short tasks that run many iterations. When running these applications on a parallel processing framework like Apache Spark, one problem that affects the running time is the straggler, where a disproportionate long-running task slows down the entire cluster. In this work we present a straggler mitigation technique tailored for applications that run small tasks for many iterations over a large dataset, and implemented the algorithm in Apache Spark. We monitor the resources available on each Spark node, and dynamically re-partition the dataset proportional to the estimated resource available. We have shown that our algorithm has negligible overhead for resource monitoring, and can improve the performance of Spark cluster significantly when stragglers are present.

# Acknowledgements

This thesis would not have been possible without the support of many people. I would like to thank my advisor, Professor Roy H Campbell, who offered great guidance throughout the entire process. Also, thanks to Abdollahian Noghabi, Shadi, who generously provided help in solving difficulties I met constructing this thesis. I would also like to show my gratitude to Mary Beth Kelley from the CS advising office for her dedicated guidance in editing my thesis. In addition, I am immensely grateful to the University of Illinois Graduate College and Department of Computer Science for offering me a teaching assistant position to help me complete the degree in Master of Science. Finally yet importantly, I would like to thank my parents, boyfriend, and numerous friends, without whom it would be impossible for me to reach my goals this far.

# Table of Contents

# Chapter 1

# Introduction

## 1.1 Background and Related Work

In the era of big data, data mining and machine learning has seen a rapid growth in popularity and fields of application. The datasets involved in these algorithms are also expanding swiftly in terms of size and complexity. Apache Spark is a fault-tolerant parallel computing engine that has gained increasing popularity. With its multi-language supported API, easy to use machine learning library MLLib, and its efficiency edge in iterative computation, Spark is well suited for machine learning applications [1]. Many of the popular data science algorithms, such as K-Means and PageRank, feature relatively short computation that run in hundreds of thousands of iterations. The major cost of each iteration is the repetitive computation on all entries of the dataset. With the available computation resource fixed, running time of each iteration of many iterative algorithms is largely proportional to the size of the dataset.

One common issue that hurts the performance of parallel computation engine like Apache Spark is the presence of stragglers, where a disproportionate long-running task slows down the entire cluster. When computation reach a checkpoint, results are gathered from each executor node, and the next step of calculation can only start after results from all executor nodes are returned. Therefore the slowest running node controls the running time of a Spark application, and the task that slows down the entire application is called a straggler.

Current straggler mitigation techniques in distributed computation engines can be broadly classified into two categories: speculation and blacklisting. For speculation, a task is speculated to be a straggler on a timeout, and is then replicated on other nodes. The result of the first finished copy is used for further calculation. In blacklisting, stragglers are mitigated by blacklisting unhealthy nodes for certain amount of time.

Multiple straggler mitigation solutions proposed fall into the category of speculation. Zaharia, Matei, et al proposed a system, LATE that make improvements for stragglers in MapReduce related systems. The key idea of LATE scheduler is to rank all non-speculated running tasks and assign a copy of the estimated slowest task to the next available machine [2]. Ananthanarayanan, Ganesh et al introduced a cause-aware task monitor system, Mantri that can restart or duplicate outlier tasks [3]. However both systems suffer from short-sightedness for iterative applications: with straggler spans many iterations, a delay of timeout and task duplication must be applied to each iteration.

Ananthanarayanan, Ganesh et. al. took the speculation to extreme by launching multiple clones of the same task to deal with stragglers in system running small jobs . Using this technique on small jobs can increase the utility of a Spark cluster. The cloning technique works under system when the smallest 90% of the jobs only consume 6% of the resources in execution [4]. This does not work under our problem setting where all computation resource consumption peaks at the same time for each iteration.

Spark's current straggler mitigation technique is based on speculation. When a fraction of tasks has successfully returned, Spark waits for a multiple of the median completed task running time. When the timeout threshold is reached, any unfinished tasks are speculated as a stragglers and duplicated [5]. With Spark's default straggler mitigation strategy, a job could theoretically be slowed down by up to 150% without task duplication triggered. Iterative jobs may slow down more when duplication is applied to each iteration that has stragglers.

Another straggler mitigation technique in distributed computing engines is blacklisting. Blacklisting finds nodes in bad condition and blocks task scheduling on these machines for a certain amount of time. Facebook and Bing clusters, for example, blocks 10% of their machines [4]. However, straggling can happen on nodes that are not blacklisted due to complex reasons [4], and a node may recover to good condition long before the blacklisting timeout, leading to waste of resources.

Neither speculation nor blacklisting is a satisfactory solution for alleviating stragglers in iterative applications. With the increasing demand and popularity in iterative machine learning applications, there is a need to develop a better straggler mitigation technique for those applications.

## 1.2 Problem Statement

Parallel data processing engines like Apache Spark suffer from disproportionate long-running tasks called stragglers. Because the slowest task in a distributed system controls the speed of the cluster, stragglers can significantly slow down the computation. Stragglers are hard to deal with

because their causes may be random and complicated. There are two categories of stragglers we want to deal with:

1. Stragglers that happen randomly in a large system. Straggling tasks can occur randomly in large systems due to complicated reasons depending on machine characteristics (ex. Disk failure, CPU scheduling limitation and memory availability), network characteristics (ex. Packet drop) and execution environment (ex. Improper task scheduling) [4] .
2. Persistent stragglers occur because of a heterogeneous environment (hardware or software). On one hand, upgrading machine clusters or datacenters usually result in a mix of heterogeneous machines [6]. This would result in applications running in heterogeneous environment with different disk speed, memory availability and CPU speed. On the other hand, software environment like running executor and driver on the same node would also lead to heterogeneity in computation resources.

Furthermore, current straggler mitigation strategies are not well suited for applications that run short tasks for many iterations. In this work we implemented an algorithm supplementing Spark's straggler mitigation technique to improve the application running time for iterative applications with short tasks in straggler present environment.

## 1.3 The Data Re-Partition Approach

In this work we present a straggler mitigation technique that monitors resources on each node and dynamically re-partitions the dataset on each node according to the resource available. Because the size of dataset and therefore amount of computation is proportional to the estimated

resources, iteration time for tasks on all nodes are comparable even during the period of resource imbalance.

## 1.4 Thesis Organization

In Chapter 2 we provide design and implementation details for computation resource monitoring and data re-partitioning with weight. Chapter 3 introduces parameter tuning for the APIs and how to avoid potential problems of thrashing. Chapter 4 shows the experimental result for algorithm overhead and performance improvement under straggler simulation.

# Chapter 2

## Design and Implementation

### 2.1 Algorithm Overview

In this work we present an algorithm that mitigates straggler for applications featuring short, iterative tasks over large dataset, and implemented the algorithm in Apache Spark. In our algorithm we monitor the computation resources on each node by measuring task run time. When resource imbalance is detected in a cluster of nodes, data is re-partitioned in the cluster, such that the size of the data partition on each node is proportional to the estimated computation resource available for that node. Because Spark schedules a task for each data partition, the computation size on one node is therefore proportional to the resource on that node. In Spark, we provided two user level APIs for spark users to use. Design and implementation details are presented in Section 2.2 to Section 2.4. When running iterative applications with repetitive short tasks over large dataset, our algorithm is applicable for two types of stragglers mentioned in Section I.2:

1. For straggling tasks that happen randomly, we re-partition dataset by weight proportional to the node's resources after resource imbalance is detected, with some heuristics to control re-partition frequency and triggering conditions. We talk about re-partition frequency and triggering condition tuning in Chapter 4.
2. For persistent stragglers in a heterogeneous environment, our algorithm re-partitions data at the beginning of the application run. User can choose either to partition data manually based on their knowledge of the system before executing any tasks, or simply let our algorithm automatically detect the resource imbalance and re-partition during execution. For persistent stragglers in heterogeneous environment, re-partitioning with weight

proportional to node resource improves the performance during entire application running time.

## 2.2 API Design Considerations

Each Spark Application differs in many aspects, including cluster size, cluster hardware, network condition, dataset size, application iterations, iteration duration and computation complexity etc. These factors affect the cost and performance gain of dataset re-partitioning by weight when stragglers are present. Cluster size, network condition and dataset size directly affects the cost of re-partitioning data across different nodes. Application iterations, iteration duration and computation complexity affects the decision of whether data re-partition is worthwhile considering cost and possible performance gain. All these aspects should be considered when determining whether data re-partitioning should be triggered.

For this reason it is impractical to hard code the automatic computation resource monitoring and data re-partitioning algorithm into Spark, but rather, we should allow users to apply different heuristics to specific application. For example, for applications that run extremely short iterations, one iteration of calculation may include large common overhead, and the returned resource estimation should be adjusted to exclude the common overhead. For clusters with known high fluctuation the users may want to accumulate the returned estimate of computation resource over many iterations and take the average. Therefore, we expose the resource monitoring and data re-partitioning APIs to Spark users, so that parameters affecting re-partition frequency and triggering conditions can be chosen on application level.

| *SparkContext.getWeightMap(partitionGranularity, prevLocWeight, taskSets)* |
|---|
| Returns a tuple of (IP-to-iteration-duration map, IP-to-partition-weight map) with type *(HashMap[String, Long], HashMap[String, Int])*<br>Parameters<br>*partitonGranularity: Double*<br>Approximate sum of weight in the returned IP to partition weight map<br>*prevLocWeight: HashMap[String, Int]*<br>IP to partition weight map used in previous iteration<br>*taskSets: Array[Int]*<br>Task sets to measure duration for counting from the last executed task set. |
| *RDD[T].repartitionWithWeight(locWeight: HashMap[String, Int])* |
| Returns a new RDD(Resilient Distributed Dataset) partitioned on each node by weight specified<br>Parameters<br>*locWeight: HashMap[String, Int]*<br>IP to weight map that indicates the weight of data on each IP address |

Table 1 - Details for *SparkContext.getWeightMap* and *RDD[T].repartitionWithWeight* API

Specifically, we provide two user-level APIs, *getWeightMap* and *repartitionWithWeight*. The *getWeightMap* is implemented in the *SparkContext* class, and is called at end of an iteration to estimate the computation resource available on each node based on the task running time. The *repartitionWithWeight* is implemented in the *RDD[T]* class, and is called to perform data repartitioning with a specified weight on each node. Details of the two Spark APIs are shown in Table 1.

## 2.3 Resource Monitoring

The first step of our algorithm is to monitor the system resource fluctuations in the system. We monitor the computation resource on each node heuristically using task computation time. Also, because the total computation time is proportional to the size of the dataset in our problem setting, the estimated resource need to be normalized by the size of the data on that executor node. Specifically, we approximate the computation resource using the inverse of task computation duration divided by the current weight on that node:

$$R_i = \frac{1}{T_i} / w_i \quad (1)$$

where $R_i$ is the estimated relative resource on node $i$, $T_i$ is the task running time on node $i$, and $w_i$ is the current weight of node $i$. In the first iteration all nodes starts with a weight of 1.

Then the percentage of resource on one node is calculated by dividing $R_i$ with the sum of estimated resources in the cluster:

$$P_i = R_i / \sum_{k=i}^{n} R_k \quad (2)$$

where $P_i$ is the percentage of resource on node $i$, $R_i$ is the estimated relative resource on node $i$, and $\sum_{k=i}^{n} R_k$ is the summation of all estimated relative resources in a cluster of n nodes. Note that the duration does not include the network delay to fetch the result, because we're only interested in computation time. For this reason, our algorithm does not work for stragglers due to faulty network only.

The *getWeightMap* API takes in a parameter, *partitionGranularity*, which is the approximate sum of weights of the returned IP-to-weight map. The new weight for each node is used later for data partitioning, and is calculated by multiplying $P_i$ with *patitionGranularity*:

$$w_{i,new} = P_i \times g \tag{3}$$

where $w_{i,new}$ is the new weight to be used for future iterations, $P_i$ is the estimated percentage of resource on node $i$, and $g$ is the *partitionGranularity* from API parameter.

The actual sum of weights returned is not exactly equal to *partitionGranularity*. This is because of rounding in weight calculation. The partitionGranularity affects re-partition cost and expected performance gain. Details can be found in Chapter 4, where we talk about parameter tuning. If the users want to apply different heuristics to get the weight for each node, we also return an IP-to-duration-map for the users to use as a reference. Implementation details for the *getWeightMap* can be seen in Algorithm 1

```
def getWeightMap(granularity, prevLocWeight, taskSets)
{
 initialize durationMap
 initialize weightMap
 sumInverse = 0.0

 foreach ts in taskSets
 {
  fetch ts.taskInfos
  foreach key is taskInfos.keys
  {
   host = taskInfos[key].host
        duration = taskInfos[key].duration- taskInfos[key].getRemoteFetchTime
        sumInverse += 1/(duration/prevLocWeight[host])
        durationMap += (host->duration)
  }
 }

 foreach e in durationMap
 {
  percentPower = 1/(e.value/prevLocWeight[e.key], 1))/sumInverse
  weightMap += (e.key->math.round(granularity*computationPower))
 }

 return (durationMap, weightMap)
}
```

Algorithm 1 - Get IP- to-Weight Map

## 2.4 Re-partition Resilient Distributed Dataset with Weight

The second step in our algorithm is to re-partition the data by specified weight once resource imbalance is detected. Spark dataset is represented as Resilient Distributed DataSets (RDDs). In Spark's RDD[T] class, we provide an API, *repartitionWithWeight* to re-partition the RDD to each node with different weight, where the weight should be proportional to the node's estimated available resource. The *RDD[T].repartitionWithWeight* takes an IP-to-weight map, with keys $loc_i$ being the IP address of node i and values $wt_i$ being the weight corresponding to that location. Data re-partitioning with weight includes two steps: re-partition and local coalesce. Figure 1 illustrates the two steps when calling *repartitionWithWeight(HashMap[node1->3, node2->2])*, and implementation details of the API *SparkContext.getWeightMap* is shown in Algorithm 2

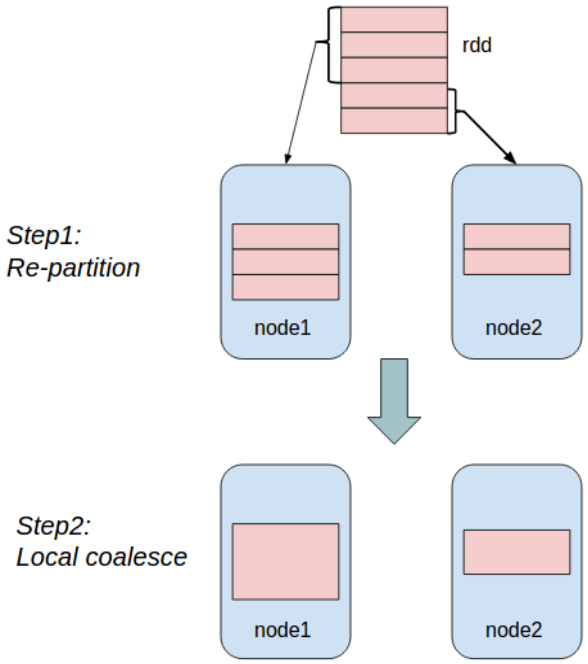


Figure 1 - Re-partition RDD with Weight Steps

The first step of re-partitioning the RDD by weight is to partition RDD into sum_weight number of partitions and assign them to different partition groups at different nodes. The sum_weight is calculated as:

$$\sum_{i=0}^{n} w_i \qquad (4)$$

where $w_i$ is the weight on node $i$. For each partition group at node $i$, we assign $w_i$ number of partitions to that partition group. Data transferring over network is expensive, therefore when assigning partitions to partition groups, we try to move data around as little as possible by assigning partitions with locality preference first if possible. A partition's preferred location is the node where a copy of that partition already exists. The sum_weight number of partitions are grouped into two arrays, *partsWithLocs* (partitions with preferred locations) and *partsWithoutLocs* (partitions without preferred locations). Figure 1 illustrates the initial state of partition groups initialized with location 1, 2, 3, 4 and weight 1, 3, 3, 4, partsWithLocs initialized with preferred locations 1, 1, 2, 2, 3 and partsWithourLocs.

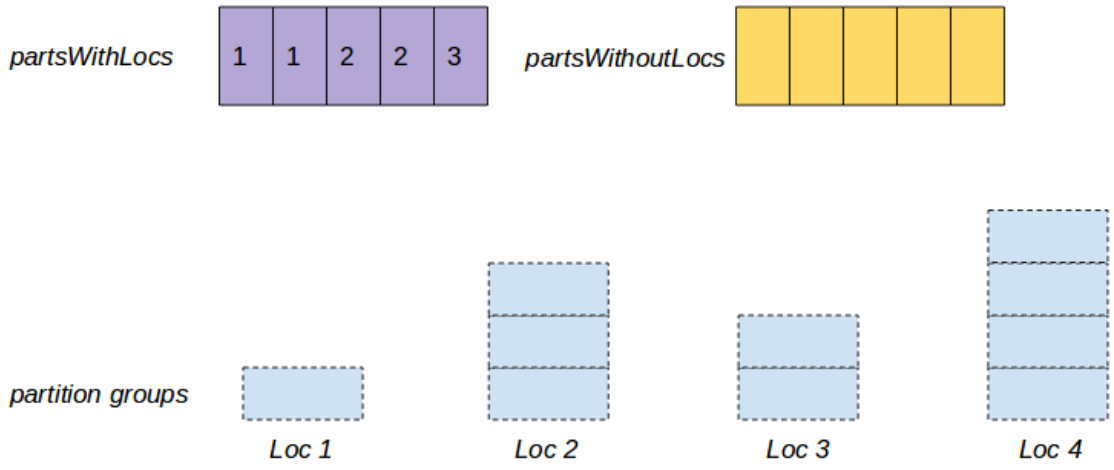


Figure 2 - Initial States of Partition Groups, *partsWithLocs* and *partsWithoutLocs*

First, we round robin through the array *partsWithLocs*, and assign each partition to its desired location if possible. If a partition's desired $loc_i$ already have $wt_i$ number of of partitions assigned, then the partition's locality is ignored and is added to the *partsWithoutLocs* array. Figure 3 illustrates assigning partitions with preferred locations. Note that the second partition in *partsWithLocs* failed to be assigned to its prefered location of 1, and is added to the *partsWithoutLocs* array. Its data locality is ignored and is later assigned to a partition group as a partition without location preferences.

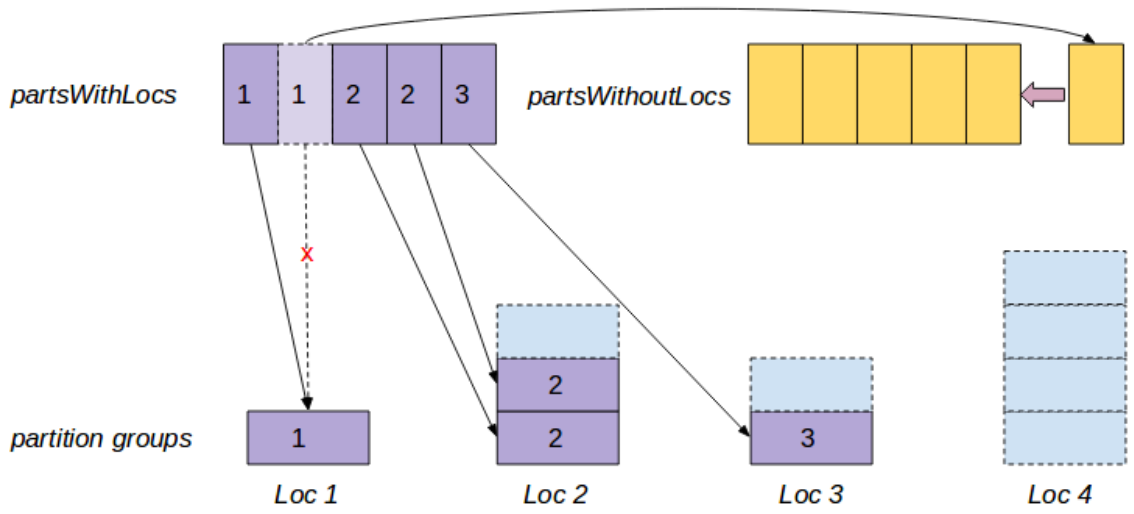


Figure 3 - Assigning Parts with Preferred Locations

After all elements in *partsWithLocs* are traversed once, no other location preferences of any partition can be fulfilled. We then round robin through *partsWithoutLocs* array and assign the remaining partitions to the first partition group with slot available, until all partitions are assigned. Figure 4 illustrates assigning partitions without preferred locations.

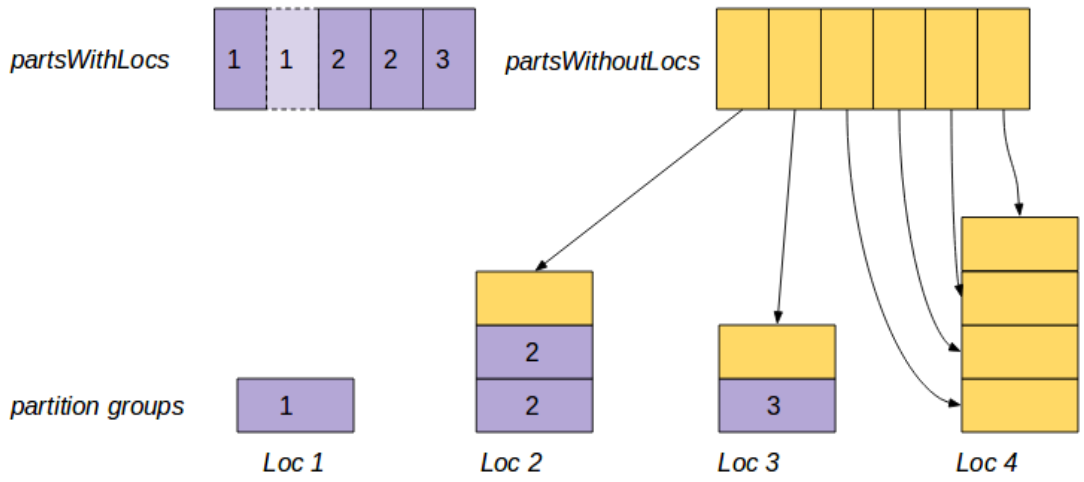


Figure 4 - Assigning Parts without Locations

The second step is to coalesce the partitions allocated to one location into a single partition. Spark tasks are scheduled for each partition, and there is a scheduling delay and launch overhead involved to execute each task. Therefore, we coalesce all partitions assigned to each node into one single partition to minimize the overhead. Algorithm 2 shows the implementation details of the *repartitionWithWeigtht* API.

```
def coalesceWithWeight(parentRDD, locWeight)
{
 sumWeight = locWeight.values.sum
 divide RDD into sumWeight partitions
 Initialize parittionGroups
 foreach loc in locWeight.keys:  initialize partitionGroups[loc]

 partsWithLocs += (partition, prefLoc)
 partsWithoutLocs += (partition)
 foreach (part, locs) in partsWithLocs
 {
        assigned = false
        foreach loc in locs
        {
                if partitionGroups[loc].size < locWeight[loc]
                {
                        partitionGroups[loc] += part
                        assigned = true
                        break
                }
                if (assigned == false)  partsWithoutLocs += part
        }
        iter = partsWithoutLocs.iterator
        foreach group in partitionGroups
        {
                 while(group.size < locWeight[group.key]) group += iter.next()
        }
        coalesce partitions into one partition at each location
        return partitionGroups
 }
```

Algorithm 2 - Re-partition RDD with Weigh

When using the *repartitionWithWeigtht* API, user should also decide the triggering condition of and frequency of data re-partitioning. Detail regarding re-partition frequency tuning is presented in the next chapter.

# Chapter 3

## Parameter Tuning

### 3.1 API Parameters

To use the API to improve the cluster performance, several parameters should be tuned. The *getWeightMap* API takes in *partitionGranularity*, the approximate sum of weights for each nodes. The larger the *partitionGranularity*, the more fine-grained duration difference will be captured by the weights. However, because approximating computation resource of node from task duration has an error built in, too large of a *partitionGranularity* is not necessary.. Another parameter for *getWeightMap* is *taskSets*. A task set in Spark is a set of same tasks executed on different partitions of a RDD, and tasks in a task set share the same task id. The parameter *taskSets* specifies task sets whose executing durations we're interested in. For each iteration Spark executes several task sets, user of the API should select task sets that involves computation over the entire dataset to get an accurate estimate of optimal IP-to-weight map.

### 3.2 Other Parameters

Apart from the parameters in the APIs, user should also select a criteria to control the triggering condition and frequency of re-partitioning when using the APIs. As an example we've modified Spark's MLLib K-Means algorithm to implement our algorithm. We used *durationRatioLimit* to control the triggering condition and *ephemeralLimit* to control repetition frequency. Specifically, *durationRatioLimit* is the upper limit of the allowed min to max duration ratio measured, for tasks in a taskset executed on different node. If the duration ratio is higher than the threshold specified by durationRatioLimit, *repartionWithWeight* is triggered. A high durationRatioLimit

would promote the algorithm to reach better performance. In our modification of Spark's KMeans, we used a default *durationRationLimit* of 0.7.

The ephemeralLimit is the minimum number of consecutive iterations required for the duration ratio to be lower than the durationRatioLimit. When the duration ratio is lower than the durationRatioLimit for ephemeralLimit number of consecutive iterations, the algorithm would have more confidence that the imbalance in computation resource is not a transient phenomenon, and re-partition the data by weight. In our modification of Spark's K-Means, we used an ephemeral limit of 3, and the weight for each node was averaged among the three consecutive resource imbalance iterations.

## 3.3 Parameter Tuning and Thrashing

One problem that may significantly hurt the performance of our algorithm in a high fluctuation system is thrashing, where estimated resource imbalance fluctuates among nodes and data partitioning happens at a high frequency. Parameters should be carefully tuned to avoid thrashing heuristically.

The *ephemeralLimit* is one parameter used to prevent thrashing. With *ephemeralLimit* set to larger than one, we filter out transient system variations. As the number of consecutive iterations of resource imbalance approaches the *ephemeralLimit*, there is more statistical confidence that the imbalance is not ephemeral.

Another parameter that affects thrashing is the *dutationRatioLimit*. A high *durationRatioLimit* would promote the system to converge to a more reasonable data partition. However, because of the built-in error of using execution time to estimate available resource, and the unavoidable system fluctuation, too high of a *durationRatioLimit* would lead to thrashing. Therefore we should allow some duration difference to allow for minor system volatility and resource estimate errors. Also note that, a high durationRatioLimit should have a corresponding high partitionGranularity. Otherwise thrashing may happen when the system strives to reach the *durationRatioLimit* by re-partitioning the data, while the partition is too coarse grained to capture the duration difference. In our experiments we used 4 to 6 multiples of number of executors and a durationRatioLimit of 0.7. When selecting the partitionGranularity and durationRationLimit, re-partition cost and performance gain expectation should both be considered.

# Chapter 4

# Evaluation

## 4.1 Evaluation Environment Setup

Our evaluation is performed on an Amazon EC2 cluster of 7 c4.xlarge machines, each with 4 virtual CPUs, 7.5 GB memory and EBS block storage. One machine is used as the driver node that runs the Spark master, and the other six machines are used as executor nodes running Spark slaves. We simulated stragglers by setting the CPU limit of the Spark executor. All experiments presented in this Chapter uses the same experiment setup mentioned above.

## 4.2 Cost of Re-partition

Data re-partitioning requires chunks of data to be transferred over the network, and therefore have associated costs. We examine the cost of re-partitioning on datasets of 50MB, 100MB, 500MB and 1000MB, and compare the re-partitioning cost with the minimal computation cost of one iteration. We selected counting the dataset as the minimal computation on the dataset, and used the extreme case of moving the entire partition of one executor to the other five executor nodes. It is worth noting that in most applications computation over the dataset is much more complicated than counting, therefore the ratio of re-partition cost to computation cost should be smaller than what we show in our conservative experiments.

Figure 5 shows the count computation cost of one iteration in seconds, re-partitioning cost in seconds and the cost ratio. From the cost ratio series we can see that even in the extreme case re-partition costs less than 3 iterations of computation time. In real applications where computation is much more complicated than counting, the cost ratio should be lower.


Figure 5- Re-partition Cost vs. Min Computation Cost

## 4.3 Resource Monitoring Overhead

To understand the cost of resource monitoring, we ran experiments on real datasets and compared the running time with that of the Spark default.

### 4.3.1 Experiment Dataset

We run the K-Means algorithm on 0.3 million New York Times article downloaded from the UIC Machine Learning Repository. The dataset contains 102660 vocabulary and total of 69679427 words, with stop words removed and vocabulary truncated by keeping only words that occurred more than 10 times. Words and document names are tokenized into unique indices. We further processed the data to represent each document as a line of dense vector, with tokenized words as indices and word counts as values. We run K-Means algorithm on the dataset to cluster

the articles into different topics, and observed performance of the system under different settings. All experiments presented in this and later sections uses this dataset.

### 4.3.2 Resource Monitoring Overhead Result

We run an experiment using default Spark MLLib K-Means algorithm and the modified version with re-partitioning monitoring enabled. We run experiments with 25, 50, 75 number of cluster centers respectively, with 100 number of iterations. Each experiment with a specific number of K-Means centers is run 10 times. The result has an average standard deviation of 3.08% for Spark default and 4.21% for resource monitoring version. We find that the overhead is 0.90%, 0.33% and 0.11% for 25, 50 and 75 K-Means centers respectively.

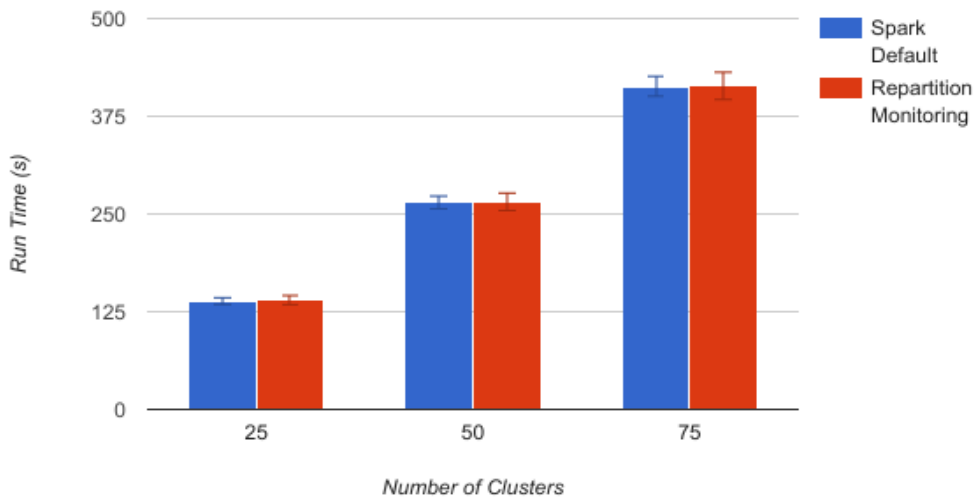


Figure 6 - Spark Default versus Resource Monitoring Run Time

Figure 6 shows the Spark default versus resource monitoring run time. From the result we can see that our resource monitoring algorithm has negligible overhead that is smaller than the average standard deviation of experiment runs.

## 4.4 Constant CPU Throttling

Stragglers can happen in heterogeneous environment, either due to hardware or software limitations. We simulate a heterogeneous environment by setting the CPU limit to 12.5, 50, and 75 percent on one executor node. We observe in a non-interfered environment the Spark executor uses up-to around 100% CPU. Theoretically, for a CPU intensive application the slowdown is up to 700%, 300% and 100% for a CPU limit of 12.5, 50 and 75 percent. In our experiment we observed a corresponding 513.54%, 218.29% and 88.27% slowdown. We run 10 experiments for each CPU limit with 25 K-Means centers for 100 iterations, and calculated the average for un-throttled and throttled environment respectively. Table 2 summarizes the results.

| CPU limit (%) | Spark Default Duration (s) | Spark Default Duration Standard Deviation (s) | Observed Slowdown (%) | Theoretical Max Slowdown (%) |
|---|---|---|---|---|
| none | 138.59 | 4.81 | - | - |
| 12.5 | 857.97 | 3.71 | 513.54 | 700.00 |
| 25 | 445.10 | 6.70 | 218.29 | 300.00 |
| 50 | 263.27 | 5.80 | 88.27 | 100.00 |

Table 2 - Spark run time for different CPU limit and slowdown

One reason for the difference between the theoretical slowdown and actual slowdown is because the 100% CPU usage of a non-throttled Spark executor is the peak and not the average. Luckily our API captures the average CPU usage heuristically by measuring the execution time of the entire computation task. Another reason for the difference between theoretical and observed slowdown is the common task overhead like scheduler delay, task deserialization time, shuffle read time, shuffle write time, result sterilization time and getting result time. We've only excluded the getting result time in our implementation, and other overheads remain a source of inaccuracy in our resource estimation algorithm. However, the overhead is small compared to the task execution time and have a small effect on our algorithm. Figure 7 shows the task time split up of one iteration of K-Means running Spark default with 25 cluster centers. From the figure we can see that miscellaneous task overhead is very small compared with the executor computing time.

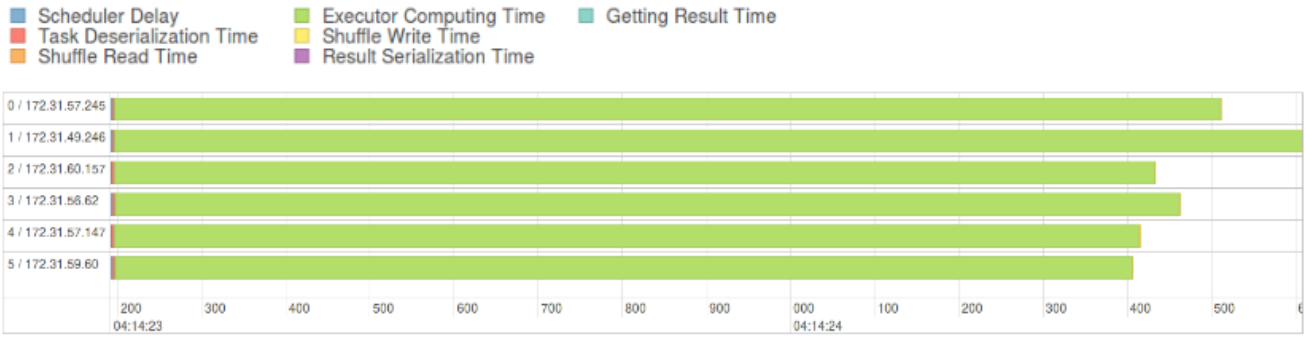


Figure 7- Spark Task Time for One Iteration of K-Means with 25 Cluster Centers

Then we run 5 experiments with dynamic re-partitioning enabled for K-Means using the same throttling setting and compared with the results of throttled Spark default. We used 36 for *paritionGranularity*, 3 for *ephemeralLimit* and 0.7 for *durationRatioLimit*. For persistent resource imbalance running with the above parameters, re-partitioning by weight happens at the third iteration of the K-Means algorithm. We can see our algorithm can mitigate up to 94% of the slowdown due to one straggler. Table 3 summarizes the comparison between Spark default versus re-partition enabled environment with one node throttled to different CPU limit, and Figure 8 shows the percent slowdown mitigated for different K-Means runs. We cannot mitigate 100% of the straggler slowdown because of the re-partition cost and relaxed re-partition heuristics.

| CPU Limit (%) | Spark Default Running Time (s) | Spark Default Running Time Standard Dev (s) | Re-partition Running Time (s) | Re-Partition Running Time Standard Dev (s) | Improvement to Spark Default (%) | Percent Slowdown Mitigation (%) |
|---|---|---|---|---|---|---|
| none | 138.59 | 4.81 | 139.84 | 6.26 | - | - |
| 12.5 | 857.97 | 9.61 | 179.71 | 3.71 | 79.05 | 94.45 |
| 25 | 445.10 | 19.15 | 175.32 | 6.70 | 60.61 | 88.38 |
| 50 | 263.27 | 42.46 | 171.92 | 5.80 | 34.70 | 74.01 |

Table 3 - Comparison of Spark Default and Re-partition Enabled Run Time

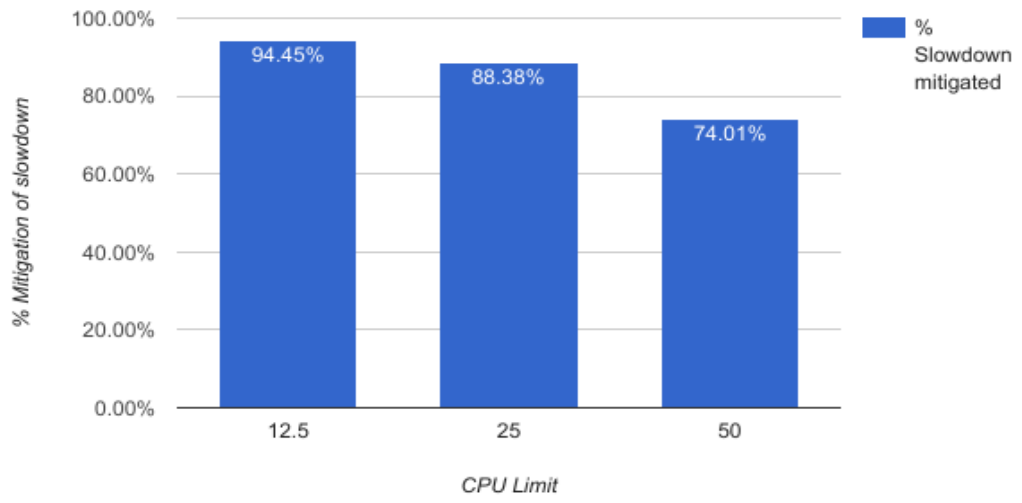


Figure 8 - Percentage of Slowdown Mitigated by Data Re-partitioning for Different CPU Limits

## 4.5 Random CPU Throttling

Stragglers can also happen randomly for complicated reasons in homogeneous clusters. We simulate this situation by randomly throttling one executor node to 50% CPU limit 25% of the time. In real world clustering problems, the number of clusters is unknown and we need to try different cluster numbers to find the best solution. Therefore a single run of our experiment consists of running K-Means with 21 to 30 number of clusters for 100 iterations each. We run experiments in both Spark default and dynamic re-partition enabled environment for 5 times respectively and calculated the average. Each single experiment runs 20-25 minutes. From the results we see that under the Spark default the throttling setting results in an average of 20.22% slow down compared to non-throttled environment. With dynamic re-partition enabled, there is a 14.28% performance improvement compared to the Spark default under throttling, which means

a 84.92% mitigation of the slowdown. Table 4 summarizes the average running time under Spark default with no throttling, Spark default with throttling and Repartition with throttling.

| | No Throttle+Spark Default (min) | Throttle+Spark Default (min) | Throttle + Re-partition (min) |
|---|---|---|---|
| Running Time | 23.0684824 | 27.73338417 | 23.77177558 |
| Standard Deviation | 0.53 | 0.85 | 1.15 |

Table 4 - Summary of runtime under different settings

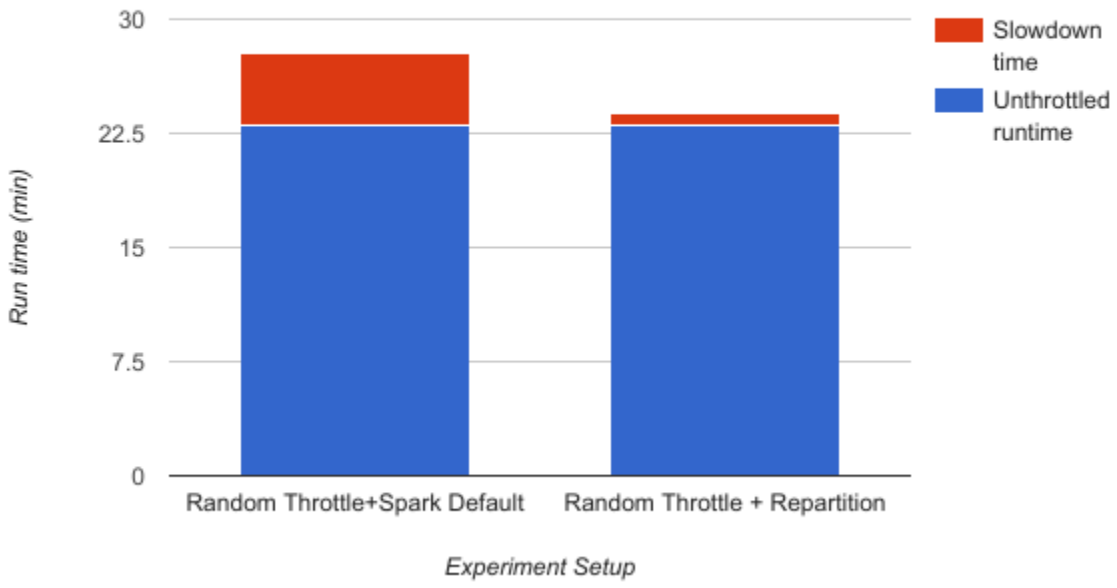


Figure 9 - Slowdown under Random Throttling for Spark Default and Re-partition Enabled Environment

Figure 9 shows the slowdown under Spark default and Reparation enabled environment with random throttling of 50% CPU limit 25% of the time. We can see the slowdown is much smaller for re-partition enabled environment than Spark default. Under this setup our algorithm can mitigate 84.92 % of the straggler slowdown. There is still a small slowdown (3%) under the re-partition enabled setting due to reduced resource, ephemeral resource imbalance identification cost, data re-partition cost and relaxed re-partition heuristics. However, the performance gain is still quite significant compared to the throttled Spark default.

# Chapter 5

## Discussion and Future Work

From our evaluation we can see our algorithm can significantly improve performance under CPU throttling for iterative applications with short, repetitive tasks. However there are several limitations to our algorithm. First, our algorithm relies on measuring task duration to estimate the resource available to each node. We excluded the time for result fetching from the task run time, but other overheads are still included in our duration measurement. These overheads include scheduler delay, task deserialization time, shuffle read time, shuffle write time and result deserialization time. Although we've shown in Section V.3 these overheads are small compared to task computation time, they remain a source of inaccuracy. In the future we will try to get a more accurate measurement of task computation time. Also, in our evaluation we did not do disk I/O throttling. However, because Spark does in-memory processing, stragglers caused by disk I/O is less of an issue compared to stragglers due to CPU limitations. Future work include a plan to evaluate our algorithm under disk I/O straggling for different algorithms. We also plan to deploy our algorithm in commercial system to observe overhead under no-straggler situations and performance improvement under straggling situations.

# Chapter 6

## Conclusions

In this work we presented an algorithm to mitigate stragglers in Spark clusters for iterative applications with short, repetitive tasks. We monitor computation resources based on task duration and detect resource imbalances. When resource imbalance is detected, we dynamically re-partition the dataset among nodes by a weight proportional to the estimated resources on nodes. This algorithm can mitigate both random stragglers and persistent stragglers. We implemented two Spark user-level APIs to monitor Spark cluster computation resources and to re-partition dataset by weight at time of resource imbalance among cluster nodes. To better improve performance and avoid thrashing, parameter must be tuned to control re-partition triggering condition and frequency.

We evaluated our algorithm using K-Means algorithm run on a Spark cluster of 7 (1 drive and 6 executors), under different CPU throttling settings. Our evaluation shows that with dynamic re-partitioning enabled, there is little-to no overhead running our algorithm when no straggler is present, but can mitigate a high percentage of slowdown when stragglers happen.

# References

[1] Xiangrui Meng et. al. MLlib: Machine Learning in Apache Spark, *Journal of Machine Learning Research* 17 1-7, 2016.

[2] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In USENIX OSDI, 2008.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In USENIX OSDI, 2010.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX OSDI*, 2013

[5] Danish Khan et. al. Empirical Study of Stragglers in Spark SQL and Spark Streaming, 2015

[6] Hongbin Yang et.al. Improving Spark performance with MPTE in heterogeneous environments. In *ICALIP*, 2016